Optimizing Native Code for Erlang

Steve Vinoski
Basho Technologies
vinoski@ieee.org
@stevevinoski

INTEGRATION, ERLANG STYLE

- External: OS processes separate from the Erlang VM
  - Ports
  - C Nodes
  - Jinterface
  - TCP/UDP/SCTP networking

INTEGRATION, ERLANG STYLE

- Internal: statically or dynamically linked into the Erlang VM
  - Erlang Built-in Functions (BIFs)
  - Port Drivers
  - Native Implemented Functions (NIFs)

INTEGRATION EXAMPLES

- **rebar** uses ports for external commands like git, grep, rsync
- Erlang's **inet_drv** port driver
  - written in C
  - supports TCP, UDP, SCTP for Erlang applications
- Riak's **eleveldb** persistence backend is a C++ NIF

NIF DETAILS

NIF DETAILS

- Start with a regular Erlang module

NIF DETAILS

- Start with a regular Erlang module
- Functions can either be stubbed out to raise errors, or have default implementations

NIF DETAILS

- Start with a regular Erlang module
- Functions can either be stubbed out to raise errors, or have default implementations
- Corresponding NIFs live in a dynamically loaded library

NIF DETAILS

- Start with a regular Erlang module
- Functions can either be stubbed out to raise errors, or have default implementations
- Corresponding NIFs live in a dynamically loaded library
- Module typically specifies a NIF loading function via -on_load

NIF DETAILS

- Start with a regular Erlang module
- Functions can either be stubbed out to raise errors, or have default implementations
- Corresponding NIFs live in a dynamically loaded library
- Module typically specifies a NIF loading function via -on_load
- NIFs replace Erlang functions of the same name/arity at module load time

NIF EXAMPLE

- Example module: **bitwise**
- Provides a function **exor/2** that takes a binary and a value
- **exor/2** computes an *exclusive or* of each byte of the binary with the argument value
- Find the code here: https://github.com/vinoski/bitwise.git

NIF EXAMPLE

```erlang
-module(bitwise).
-export([exor/2]).
-on_load(init/0).
```

NIF EXAMPLE

```
-module(bitwise).
-export([exor/2]).
-on_load(init/0).

init() ->
    SoName = filename:join(case code:priv_dir(?MODULE) of
                               {error, bad_name} ->
                                   Dir = code:which(?MODULE),
                                   filename:join([filename:dirname(Dir),
                                                  "..", "priv"]);
                               Dir ->
                                   Dir
                           end, atom_to_list(?MODULE) ++ "_nif"),
    erlang:load_nif(SoName, 0).
```

NIF EXAMPLE

```
-module(bitwise).
-export([exor/2]).
-on_load(init/0).

init() ->
    SoName = filename:join(case code:priv_dir(?MODULE) of
                               {error, bad_name} ->
                                   Dir = code:which(?MODULE),
                                   filename:join([filename:dirname(Dir),
                                                  "..", "priv"]);
                               Dir ->
                                   Dir
                           end, atom_to_list(?MODULE) ++ "_nif"),
    erlang:load_nif(SoName, 0).
```

NIF EXAMPLE

```
-module(bitwise).
-export([exor/2]).
-on_load(init/0).

init() ->
    SoName = filename:join(case code:priv_dir(?MODULE) of
                               {error, bad_name} ->
                                   Dir = code:which(?MODULE),
                                   filename:join([filename:dirname(Dir),
                                                  "..", "priv"]);
                               Dir ->
                                   Dir
                           end, atom_to_list(?MODULE) ++ "_nif"),
    erlang:load_nif(SoName, 0).

exor(Bin, Byte) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    error({nif_not_loaded, ?MODULE}).
```

NIF EXAMPLE

```
-module(bitwise).
-export([exor/2]).
-on_load(init/0).

init() ->
    SoName = filename:join(case code:priv_dir(?MODULE) of
                               {error, bad_name} ->
                                   Dir = code:which(?MODULE),
                                   filename:join([filename:dirname(Dir),
                                                  "..", "priv"]);
                               Dir ->
                                   Dir
                           end, atom_to_list(?MODULE) ++ "_nif"),
    erlang:load_nif(SoName, 0).

exor(Bin, Byte) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    error({nif_not_loaded, ?MODULE}).
```

EXOR/2 NIF

```c
static ERL_NIF_TERM
exor(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifBinary bin, outbin;
    unsigned char byte;
    unsigned val, i;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
```

```c
static ERL_NIF_TERM
exor(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifBinary bin, outbin;
    unsigned char byte;
    unsigned val, i;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
    byte = (unsigned char)val;
    enif_alloc_binary(bin.size, &outbin);
    for (i = 0; i < bin.size; i++)
        outbin.data[i] = bin.data[i] ^ byte;
    return enif_make_binary(env, &outbin);
}
```

```
static ERL_NIF_TERM
exor(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifBinary bin, outbin;
    unsigned char byte;
    unsigned val, i;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
    byte = (unsigned char)val;
    enif_alloc_binary(bin.size, &outbin);
    for (i = 0; i < bin.size; i++)
        outbin.data[i] = bin.data[i] ^ byte;
    return enif_make_binary(env, &outbin);
}
```

```c
static ERL_NIF_TERM
exor(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifBinary bin, outbin;
    unsigned char byte;
    unsigned val, i;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
    byte = (unsigned char)val;
    enif_alloc_binary(bin.size, &outbin);
    for (i = 0; i < bin.size; i++)
        outbin.data[i] = bin.data[i] ^ byte;
    return enif_make_binary(env, &outbin);
}
```

NOW FOR SOME BIG DATA

```
Eshell V6.2  (abort with ^G)
1> {ok,Bin} = file:read_file("big-data").
{ok,<<235,72,144,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,...>>}
2> byte_size(Bin).
2000000000
```

- 2 billion bytes

LET'S TIME OUR NIF

```
3> timer:tc(bitwise, exor, [Bin, 16#5A]).
{5925452,
 {<<177,18,202,90,90,90,90,90,90,90,90,90,90,90,90,90,90,
    90,90,90,90,90,90,90,90,90,...>>,
  0}}
```

LET'S TIME OUR NIF

```
3> timer:tc(bitwise, exor, [Bin, 16#5A]).
{5925452,
 {<<177,18,202,90,90,90,90,90,90,90,90,90,90,90,90,90,90,90,
    90,90,90,90,90,90,90,90,90,...>>,
  0}}
```

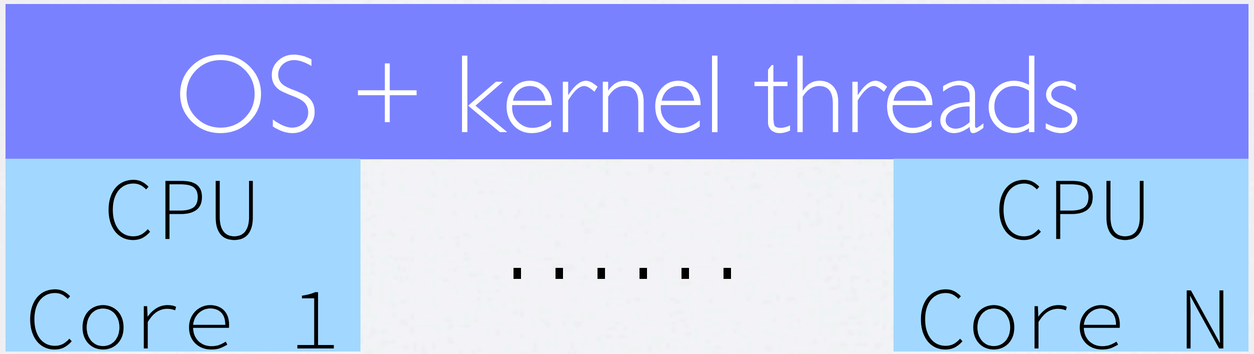
- Nearly 6 seconds!
- This is bad.

ERLANG PROCESS ARCHITECTURE

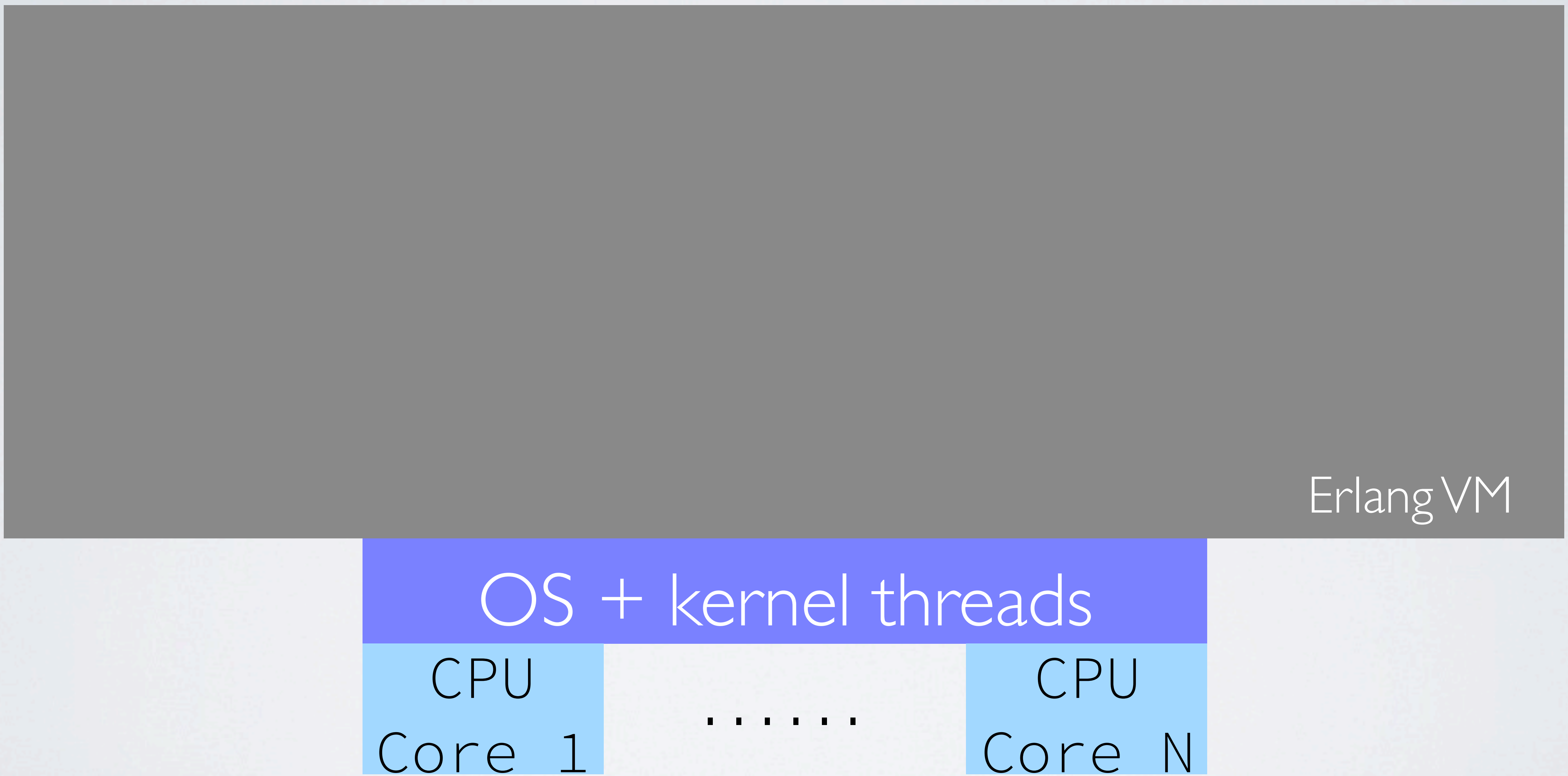
ERLANG PROCESS ARCHITECTURE

CPU Core 1	......	CPU Core N

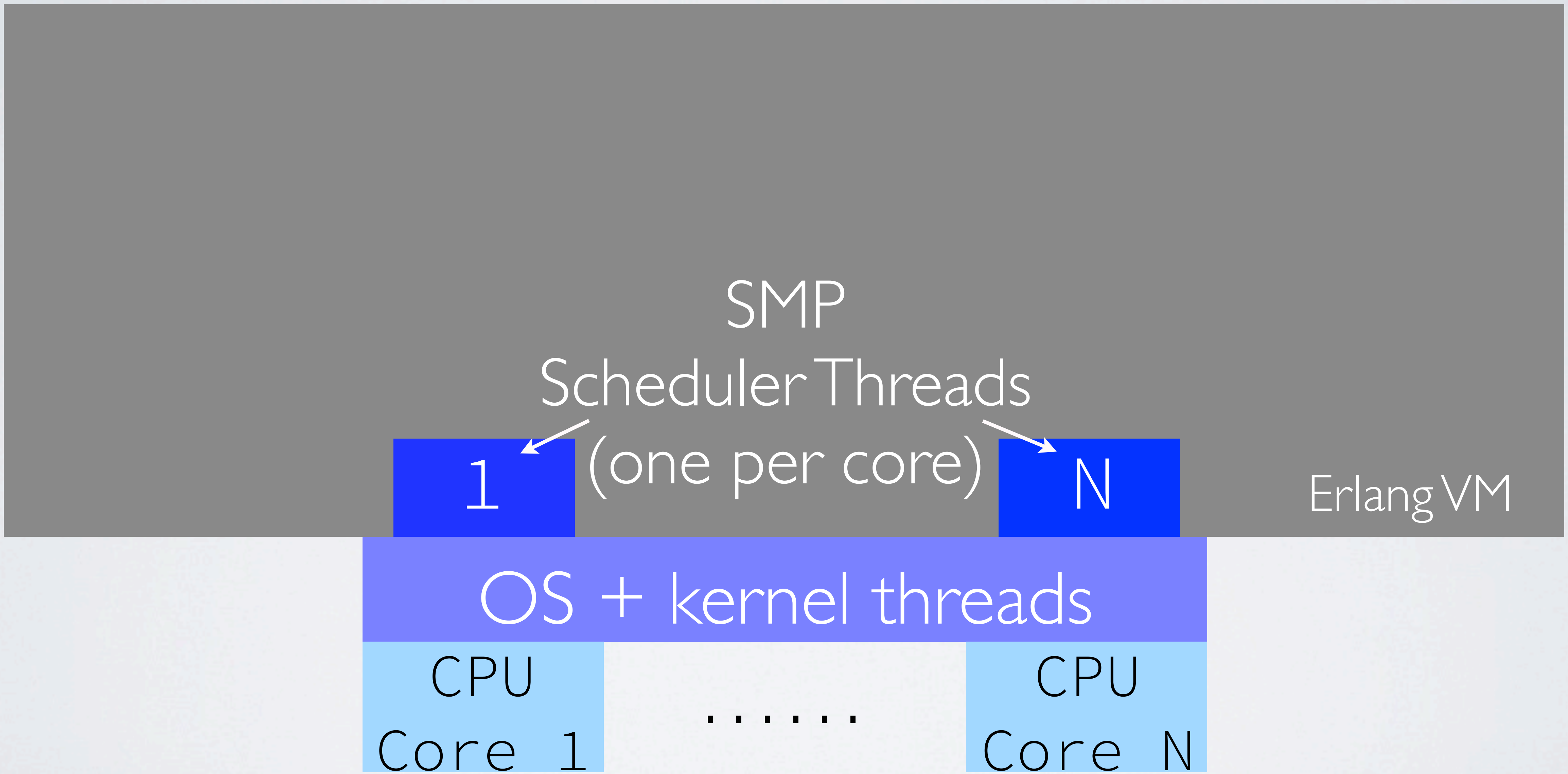
ERLANG PROCESS ARCHITECTURE



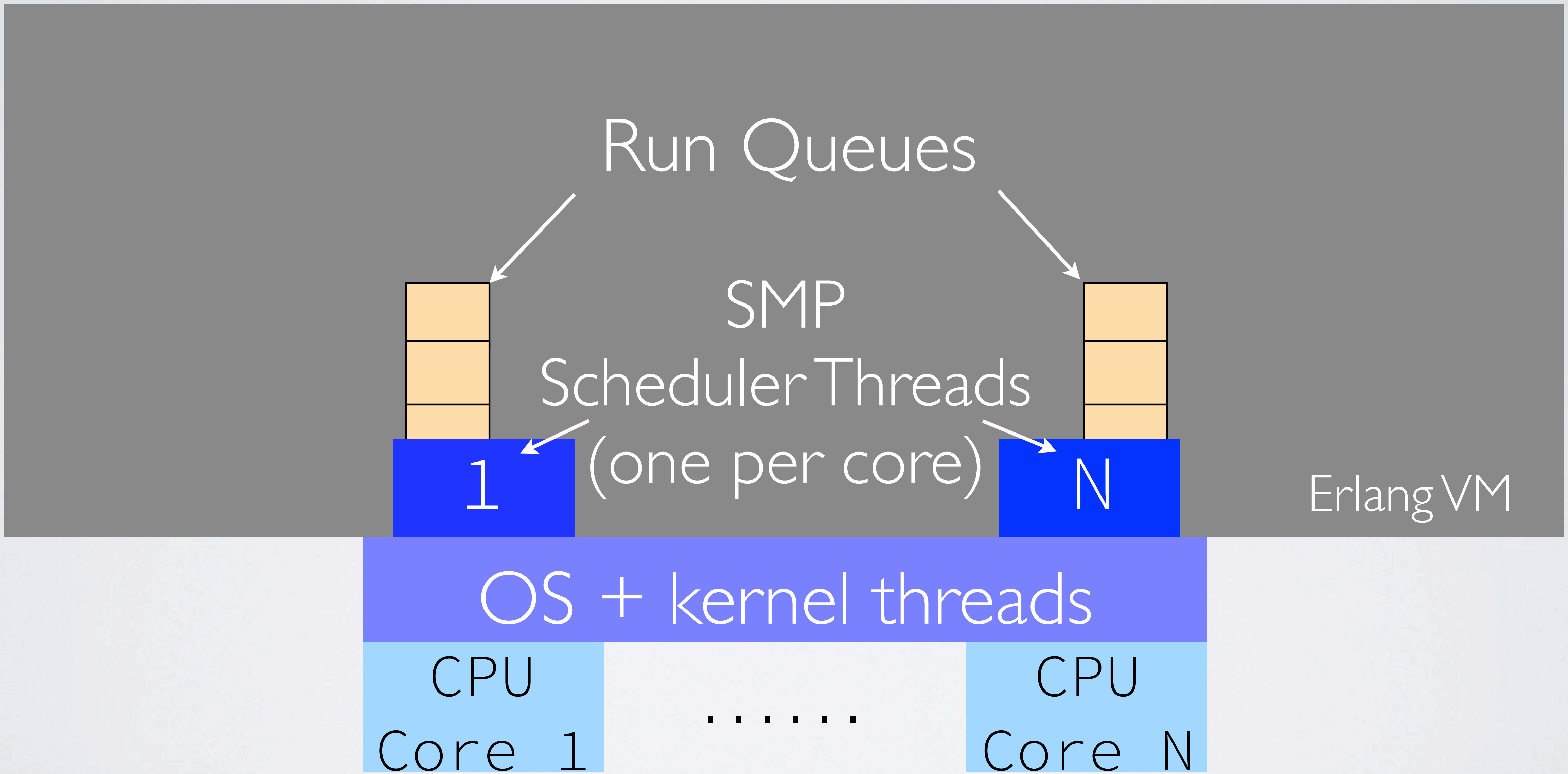
ERLANG PROCESS ARCHITECTURE



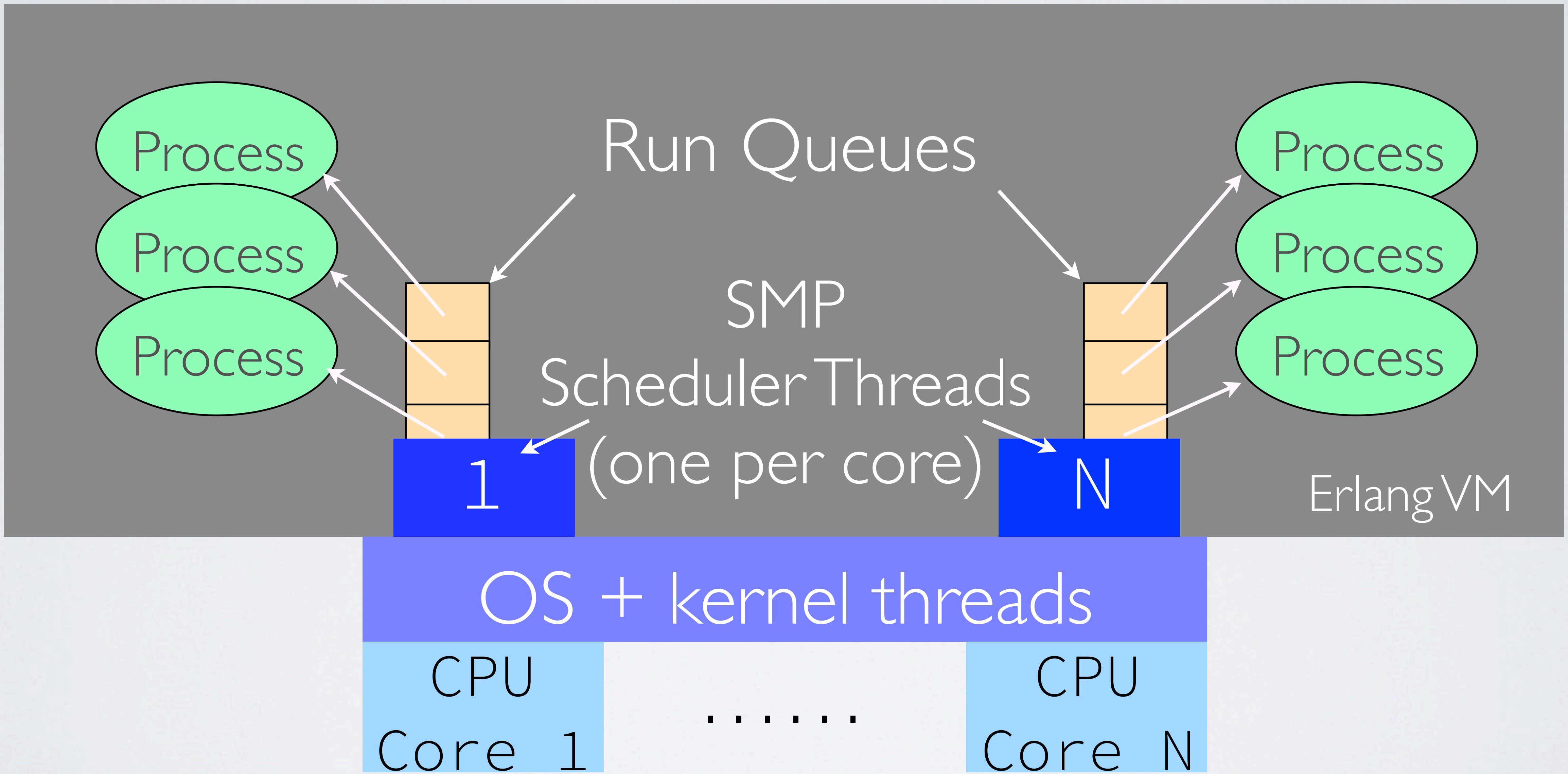
ERLANG PROCESS ARCHITECTURE



ERLANG PROCESS ARCHITECTURE



ERLANG PROCESS ARCHITECTURE



SCHEDULING A PROCESS

SCHEDULING A PROCESS

- A scheduler takes a process from its run queue

SCHEDULING A PROCESS

- A scheduler takes a process from its run queue
- It executes it until it hits 2000 reductions (function calls) or until it waits for a message, or if it hits an emulator trap

SCHEDULING A PROCESS

- A scheduler takes a process from its run queue
- It executes it until it hits 2000 reductions (function calls) or until it waits for a message, or if it hits an emulator trap
- The process then gets scheduled out and another one chosen

SCHEDULING A PROCESS

- A scheduler takes a process from its run queue
- It executes it until it hits 2000 reductions (function calls) or until it waits for a message, or if it hits an emulator trap
- The process then gets scheduled out and another one chosen
- See Jesper Louis Andersen's scheduling description: http://jlouisramblings.blogspot.com/2013/01/how-erlang-does-scheduling.html

THREAD PROGRESS

- Scheduler threads share some data structures
- But using traditional locks or ref counts to protect them scales poorly
- Instead, schedulers report their progress frequently to other schedulers
- Schedulers use their knowledge of other schedulers' progress to know when certain operations are safe
- For more details see https://github.com/erlang/otp/blob/master/erts/emulator/internal_doc/ThreadProgress.md

BLOCKED SCHEDULERS

- Blocking a scheduler prevents thread progress, making other schedulers wait
- Blocking a scheduler also makes it unavailable to run other processes
- A NIF shouldn't occupy a scheduler for more than 1-2 ms
- NIF reductions should also be counted properly

SCHEDULER COLLAPSE

- With Riak we've seen problems in production where schedulers go to sleep and stop executing processes
- Caused by misbehaving NIFs in Riak's storage backends interfering with normal scheduler operations
- Can also be caused by misbehaving standard Erlang functions
- See Scott Fritchie's **nifwait** repository, md5 branch: https://github.com/slfritchie/nifwait.git

LET'S COUNT REDUCTIONS

```
reds(Bin, Byte, Fun) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    Parent = self(),
    Pid = spawn(fun() ->
                        Self = self(),
                        Start = os:timestamp(),
                        R0 = process_info(Self, reductions),
                        {_,Yields} = Fun(Bin, Byte),
                        R1 = process_info(Self, reductions),
                        T = timer:now_diff(os:timestamp(), Start),
                        Parent ! {Self,{T, Yields, R0, R1}}
                end),
    receive
        {Pid,Result} ->
            Result
    end.
```

LET'S COUNT REDUCTIONS

```
reds(Bin, Byte, Fun) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    Parent = self(),
    Pid = spawn(fun() ->
                        Self = self(),
                        Start = os:timestamp(),
                        R0 = process_info(Self, reductions),
                        {_,Yields} = Fun(Bin, Byte),
                        R1 = process_info(Self, reductions),
                        T = timer:now_diff(os:timestamp(), Start),
                        Parent ! {Self,{T, Yields, R0, R1}}
                end),
    receive
        {Pid,Result} ->
            Result
    end.
```

A MISBEHAVING NIF

```
4> bitwise:reds(Bin,16#5A,fun bitwise:exor_bad/2).
{5857295,0,{reductions,5},{reductions,9}}
```

A MISBEHAVING NIF

```
4> bitwise:reds(Bin,16#5A,fun bitwise:exor_bad/2).
{5857295,0,{reductions,5},{reductions,9}}
```

- Blocked a scheduler thread for 5.86 seconds
- And only 4 reductions

WORKAROUNDS

- Break the data into chunks
- Call **exor_bad/2** repeatedly, once for each chunk
- Combine the resulting chunks into a final result

CHUNKING

```
exor_chunks(Bin, Byte) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    exor_chunks(Bin, Byte, 4194304, 0, <<>>).
```

CHUNKING

```
exor_chunks(Bin, Byte) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    exor_chunks(Bin, Byte, 4194304, 0, <<>>).
exor_chunks(Bin, Byte, ChunkSize, Yields, Acc) ->
    case byte_size(Bin) of
        Size when Size > ChunkSize ->
            <<Chunk:ChunkSize/binary, Rest/binary>> = Bin,
            {Res,_} = exor_bad(Chunk, Byte),
            exor_chunks(Rest, Byte, ChunkSize,
                        Yields+1, <<Acc/binary, Res/binary>>);
```

CHUNKING

```
exor_chunks(Bin, Byte) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    exor_chunks(Bin, Byte, 4194304, 0, <<>>).
exor_chunks(Bin, Byte, ChunkSize, Yields, Acc) ->
    case byte_size(Bin) of
        Size when Size > ChunkSize ->
            <<Chunk:ChunkSize/binary, Rest/binary>> = Bin,
            {Res,_} = exor_bad(Chunk, Byte),
            exor_chunks(Rest, Byte, ChunkSize,
                        Yields+1, <<Acc/binary, Res/binary>>);
        _ ->
            {Res, _} = exor_bad(Bin, Byte),
            {<<Acc/binary, Res/binary>>, Yields}
    end.
```

CHUNKING

```
exor_chunks(Bin, Byte) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    exor_chunks(Bin, Byte, 4194304, 0, <<>>).
exor_chunks(Bin, Byte, ChunkSize, Yields, Acc) ->
    case byte_size(Bin) of
        Size when Size > ChunkSize ->
            <<Chunk:ChunkSize/binary, Rest/binary>> = Bin,
            {Res,_} = exor_bad(Chunk, Byte),
            exor_chunks(Rest, Byte, ChunkSize,
                        Yields+1, <<Acc/binary, Res/binary>>);
        _ ->
            {Res, _} = exor_bad(Bin, Byte),
            {<<Acc/binary, Res/binary>>, Yields}
    end.
```

- Problem: how to determine optimal chunk size?
- Here, we arbitrarily chose 4MB chunks

CHUNKING

```
exor_chunks(Bin, Byte) when is_binary(Bin), Byte >= 0, Byte < 256 ->
    exor_chunks(Bin, Byte, 4194304, 0, <<>>).
exor_chunks(Bin, Byte, ChunkSize, Yields, Acc) ->
    case byte_size(Bin) of
        Size when Size > ChunkSize ->
            <<Chunk:ChunkSize/binary, Rest/binary>> = Bin,
            {Res,_} = exor_bad(Chunk, Byte),
            exor_chunks(Rest, Byte, ChunkSize,
                        Yields+1, <<Acc/binary, Res/binary>>);
        _ ->
            {Res, _} = exor_bad(Bin, Byte),
            {<<Acc/binary, Res/binary>>, Yields}
    end.
```

- Problem: how to determine optimal chunk size?
- Here, we arbitrarily chose 4MB chunks

CHUNKING RESULTS

```
5> bitwise:reds(Bin,16#5A,fun bitwise:exor_chunks/2).
{7869371,476,{reductions,5},{reductions,1450}}
```

CHUNKING RESULTS

```
5> bitwise:reds(Bin,16#5A,fun bitwise:exor_chunks/2).
{7869371,476,{reductions,5},{reductions,1450}}
```

- 476 chunks processed
- Much better reduction count of 1445
- Scheduler was never blocked (probably anyway)
- But a longer execution time of 7.87 seconds

A BETTER APPROACH

- For Erlang/OTP 17.3 (released 17 Sep 2014) I added a new NIF API function: **enif_schedule_nif**
- Takes a name and function pointer for a NIF, and an array of arguments to pass to it
- Schedules the argument NIF for future invocation with the specified arguments
- Allows the calling NIF to yield the scheduler

```c
static ERL_NIF_TERM
exor_yield(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifResourceType* res_type = (ErlNifResourceType*)enif_priv_data(env);
    ERL_NIF_TERM newargv[6];
    ErlNifBinary bin;
    unsigned val;
    void* res;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
```

```c
static ERL_NIF_TERM
exor_yield(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifResourceType* res_type = (ErlNifResourceType*)enif_priv_data(env);
    ERL_NIF_TERM newargv[6];
    ErlNifBinary bin;
    unsigned val;
    void* res;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
    newargv[0] = argv[0];
    newargv[1] = argv[1];
    newargv[2] = enif_make_ulong(env, 4194304);
    newargv[3] = enif_make_ulong(env, 0);
    res = enif_alloc_resource(res_type, bin.size);
    newargv[4] = enif_make_resource(env, res);
    newargv[5] = enif_make_int(env, 0);
    enif_release_resource(res);
    return enif_schedule_nif(env, "exor2", 0, exor2, 6, newargv);
}
```

```c
static ERL_NIF_TERM
exor_yield(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifResourceType* res_type = (ErlNifResourceType*)enif_priv_data(env);
    ERL_NIF_TERM newargv[6];
    ErlNifBinary bin;
    unsigned val;
    void* res;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
    newargv[0] = argv[0];
    newargv[1] = argv[1];
    newargv[2] = enif_make_ulong(env, 4194304);
    newargv[3] = enif_make_ulong(env, 0);
    res = enif_alloc_resource(res_type, bin.size);
    newargv[4] = enif_make_resource(env, res);
    newargv[5] = enif_make_int(env, 0);
    enif_release_resource(res);
    return enif_schedule_nif(env, "exor2", 0, exor2, 6, newargv);
}
```

```c
static ERL_NIF_TERM
exor_yield(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifResourceType* res_type = (ErlNifResourceType*)enif_priv_data(env);
    ERL_NIF_TERM newargv[6];
    ErlNifBinary bin;
    unsigned val;
    void* res;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
    newargv[0] = argv[0];
    newargv[1] = argv[1];
    newargv[2] = enif_make_ulong(env, 4194304);
    newargv[3] = enif_make_ulong(env, 0);
    res = enif_alloc_resource(res_type, bin.size);
    newargv[4] = enif_make_resource(env, res);
    newargv[5] = enif_make_int(env, 0);
    enif_release_resource(res);
    return enif_schedule_nif(env, "exor2", 0, exor2, 6, newargv);
}
```

```c
static ERL_NIF_TERM
exor_yield(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifResourceType* res_type = (ErlNifResourceType*)enif_priv_data(env);
    ERL_NIF_TERM newargv[6];
    ErlNifBinary bin;
    unsigned val;
    void* res;

    if (argc != 2 || !enif_inspect_binary(env, argv[0], &bin) ||
        !enif_get_uint(env, argv[1], &val) || val > 255)
        return enif_make_badarg(env);
    if (bin.size == 0)
        return argv[0];
    newargv[0] = argv[0];
    newargv[1] = argv[1];
    newargv[2] = enif_make_ulong(env, 4194304);
    newargv[3] = enif_make_ulong(env, 0);
    res = enif_alloc_resource(res_type, bin.size);
    newargv[4] = enif_make_resource(env, res);
    newargv[5] = enif_make_int(env, 0);
    enif_release_resource(res);
    return enif_schedule_nif(env, "exor2", 0, exor2, 6, newargv);
}
```

EXOR2/6

- **exor2/6** is an "internal NIF" not visible to Erlang
- Works through as much of the binary as it can before its timeslice runs out
- Reports reductions using **enif_consume_timeslice**
- When its timeslice is up, reschedules itself via **enif_schedule_nif**
- Adjusts chunksize for the next iteration based on progress in each iteration

```
static ERL_NIF_TERM
exor2(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
```

...snip...

```
    byte = (unsigned char)val;
    end = offset + max_per_slice;
    if (end > bin.size) end = bin.size;
    i = offset;
    enif_consume_timeslice(env, 0);
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
```

```c
static ERL_NIF_TERM
exor2(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
```

...snip...

```c
    byte = (unsigned char)val;
    end = offset + max_per_slice;
    if (end > bin.size) end = bin.size;
    i = offset;
    enif_consume_timeslice(env, 0);
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
```

```c
static ERL_NIF_TERM
exor2(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
```

...snip...

```c
    byte = (unsigned char)val;
    end = offset + max_per_slice;
    if (end > bin.size) end = bin.size;
    i = offset;
    enif_consume_timeslice(env, 0);
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
```

```
static ERL_NIF_TERM
exor2(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
```

...snip...

```
    byte = (unsigned char)val;
    end = offset + max_per_slice;
    if (end > bin.size) end = bin.size;
    i = offset;
    enif_consume_timeslice(env, 0);
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
```

```c
static ERL_NIF_TERM
exor2(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
```

...snip...

```c
    byte = (unsigned char)val;
    end = offset + max_per_slice;
    if (end > bin.size) end = bin.size;
    i = offset;
    enif_consume_timeslice(env, 0);
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
```

```c
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
            max_per_slice = i - offset;
            if (total > 100)
                max_per_slice -= (int)((max_per_slice*(total-100))/100);
            newargv[0] = argv[0];
            newargv[1] = argv[1];
            newargv[2] = enif_make_ulong(env, max_per_slice);
            newargv[3] = enif_make_ulong(env, i);
            newargv[4] = argv[4];
            newargv[5] = enif_make_int(env, yields+1);
            return enif_schedule_nif(env, "exor2", 0, exor2, argc, newargv);
        }
        end += max_per_slice;
        if (end > bin.size) end = bin.size;
    }
    result = enif_make_resource_binary(env, res, res, bin.size);
    return enif_make_tuple2(env, result, enif_make_int(env, yields));
}
```

```c
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
            max_per_slice = i - offset;
            if (total > 100)
                max_per_slice -= (int)((max_per_slice*(total-100))/100);
            newargv[0] = argv[0];
            newargv[1] = argv[1];
            newargv[2] = enif_make_ulong(env, max_per_slice);
            newargv[3] = enif_make_ulong(env, i);
            newargv[4] = argv[4];
            newargv[5] = enif_make_int(env, yields+1);
            return enif_schedule_nif(env, "exor2", 0, exor2, argc, newargv);
        }
        end += max_per_slice;
        if (end > bin.size) end = bin.size;
    }
    result = enif_make_resource_binary(env, res, res, bin.size);
    return enif_make_tuple2(env, result, enif_make_int(env, yields));
}
```

```
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
            max_per_slice = i - offset;
            if (total > 100)
                max_per_slice -= (int)((max_per_slice*(total-100))/100);
            newargv[0] = argv[0];
            newargv[1] = argv[1];
            newargv[2] = enif_make_ulong(env, max_per_slice);
            newargv[3] = enif_make_ulong(env, i);
            newargv[4] = argv[4];
            newargv[5] = enif_make_int(env, yields+1);
            return enif_schedule_nif(env, "exor2", 0, exor2, argc, newargv);
        }
        end += max_per_slice;
        if (end > bin.size) end = bin.size;
    }
    result = enif_make_resource_binary(env, res, res, bin.size);
    return enif_make_tuple2(env, result, enif_make_int(env, yields));
}
```

```c
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
            max_per_slice = i - offset;
            if (total > 100)
                max_per_slice -= (int)((max_per_slice*(total-100))/100);
            newargv[0] = argv[0];
            newargv[1] = argv[1];
            newargv[2] = enif_make_ulong(env, max_per_slice);
            newargv[3] = enif_make_ulong(env, i);
            newargv[4] = argv[4];
            newargv[5] = enif_make_int(env, yields+1);
            return enif_schedule_nif(env, "exor2", 0, exor2, argc, newargv);
        }
        end += max_per_slice;
        if (end > bin.size) end = bin.size;
    }
    result = enif_make_resource_binary(env, res, res, bin.size);
    return enif_make_tuple2(env, result, enif_make_int(env, yields));
}
```

```c
    while (i < bin.size) {
        gettimeofday(&start, NULL);
        do {
            ((char*)res)[i] = bin.data[i] ^ byte;
        } while (++i < end);
        if (i == bin.size) break;
        gettimeofday(&stop, NULL);
        timersub(&stop, &start, &slice);
        pct = (int)((slice.tv_sec*1000000+slice.tv_usec)/10000);
        total += pct;
        if (enif_consume_timeslice(env, pct)) {
            max_per_slice = i - offset;
            if (total > 100)
                max_per_slice -= (int)((max_per_slice*(total-100))/100);
            newargv[0] = argv[0];
            newargv[1] = argv[1];
            newargv[2] = enif_make_ulong(env, max_per_slice);
            newargv[3] = enif_make_ulong(env, i);
            newargv[4] = argv[4];
            newargv[5] = enif_make_int(env, yields+1);
            return enif_schedule_nif(env, "exor2", 0, exor2, argc, newargv);
        }
        end += max_per_slice;
        if (end > bin.size) end = bin.size;
    }
    result = enif_make_resource_binary(env, res, res, bin.size);
    return enif_make_tuple2(env, result, enif_make_int(env, yields));
}
```

A YIELDING NIF

```
6> bitwise:reds(Bin,16#5A,fun bitwise:exor_yield/2).
{5357962,5,{reductions,5},{reductions,10036}}
```

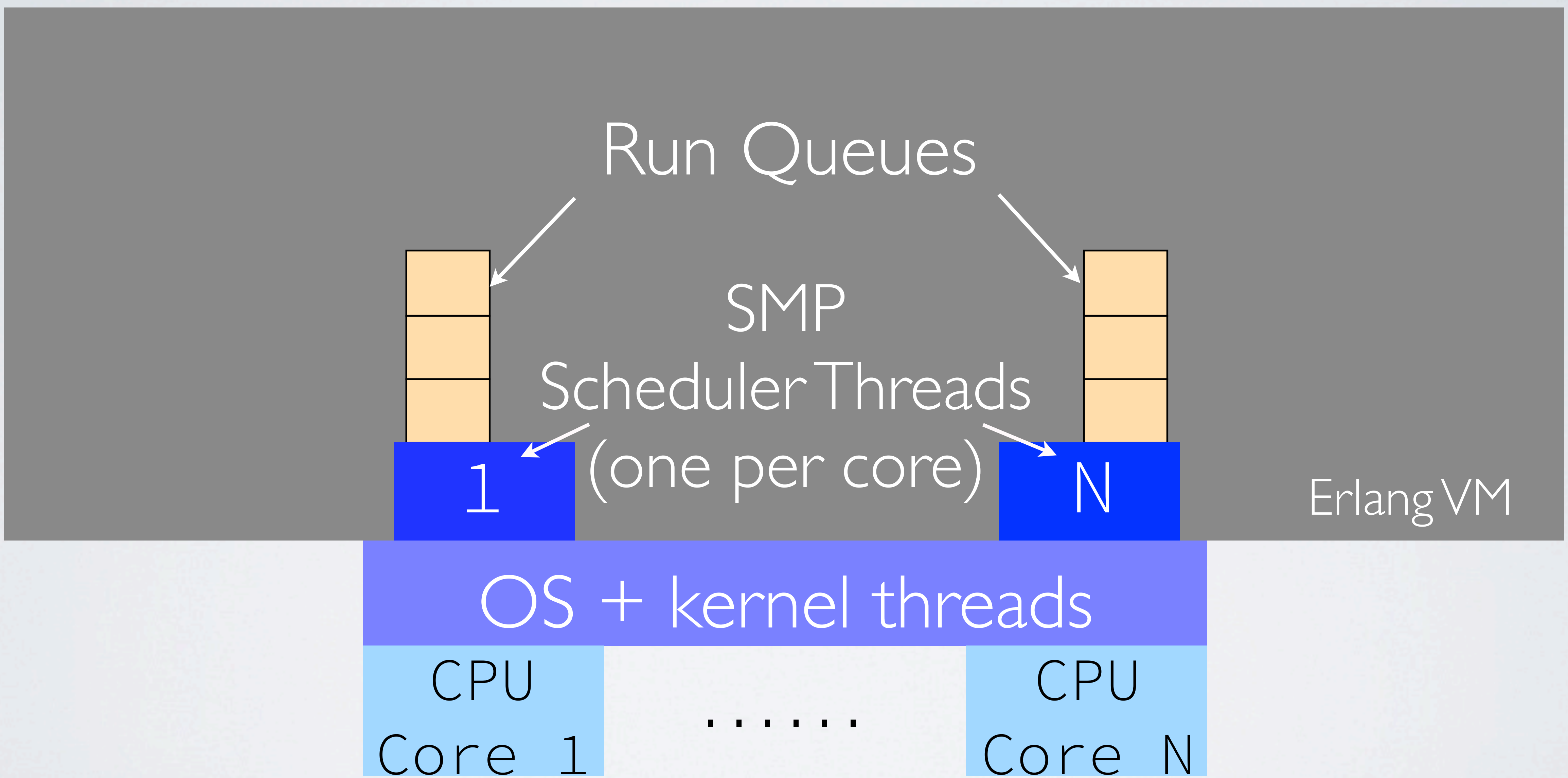
A YIELDING NIF

```
6> bitwise:reds(Bin,16#5A,fun bitwise:exor_yield/2).
{5357962,5,{reductions,5},{reductions,10036}}
```

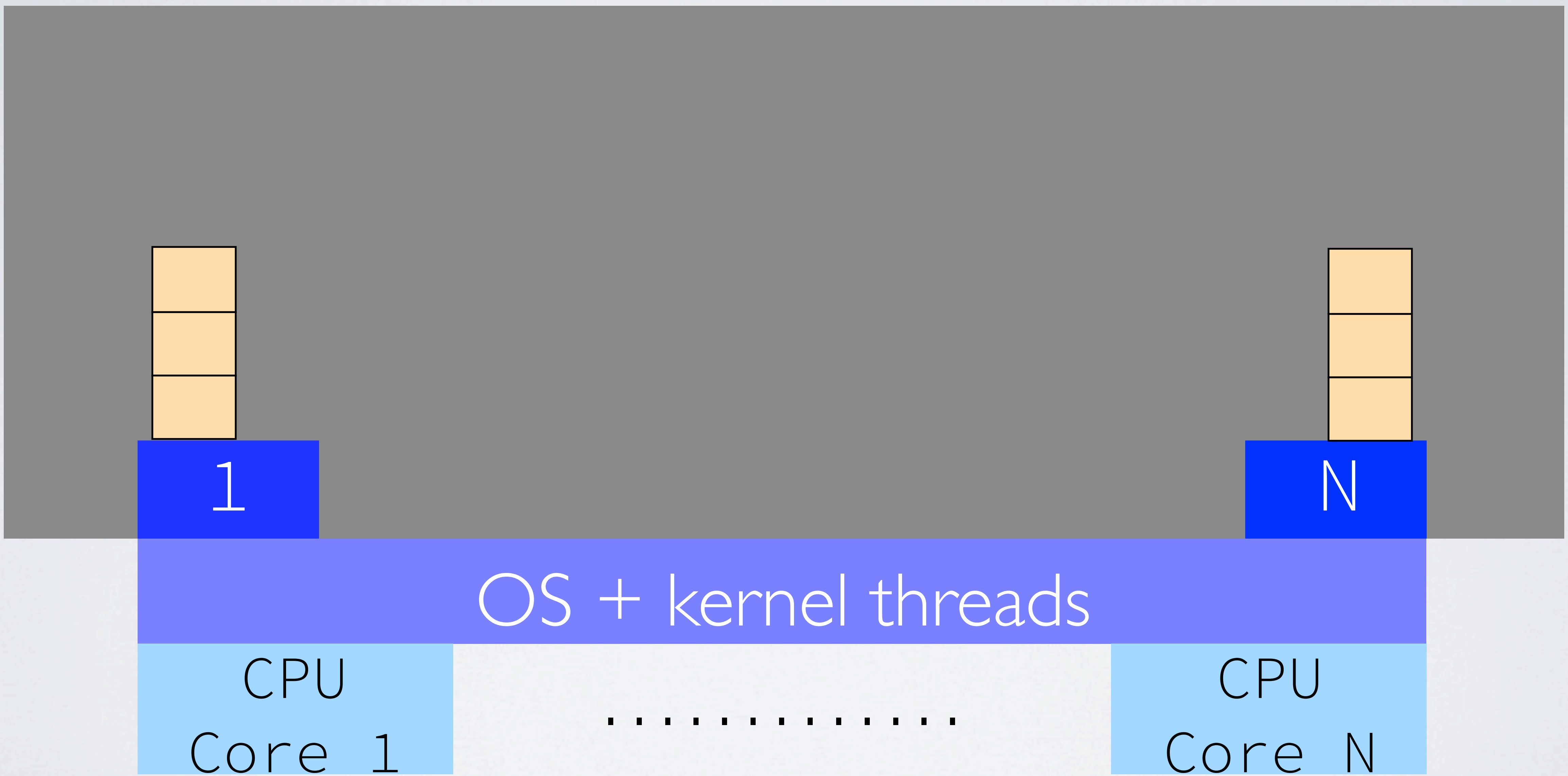
- 5.36 seconds, fastest so far
- At over 10000 reductions, much more accurate accounting
- We yielded the scheduler 5 times

ANOTHER APPROACH: DIRTY SCHEDULERS

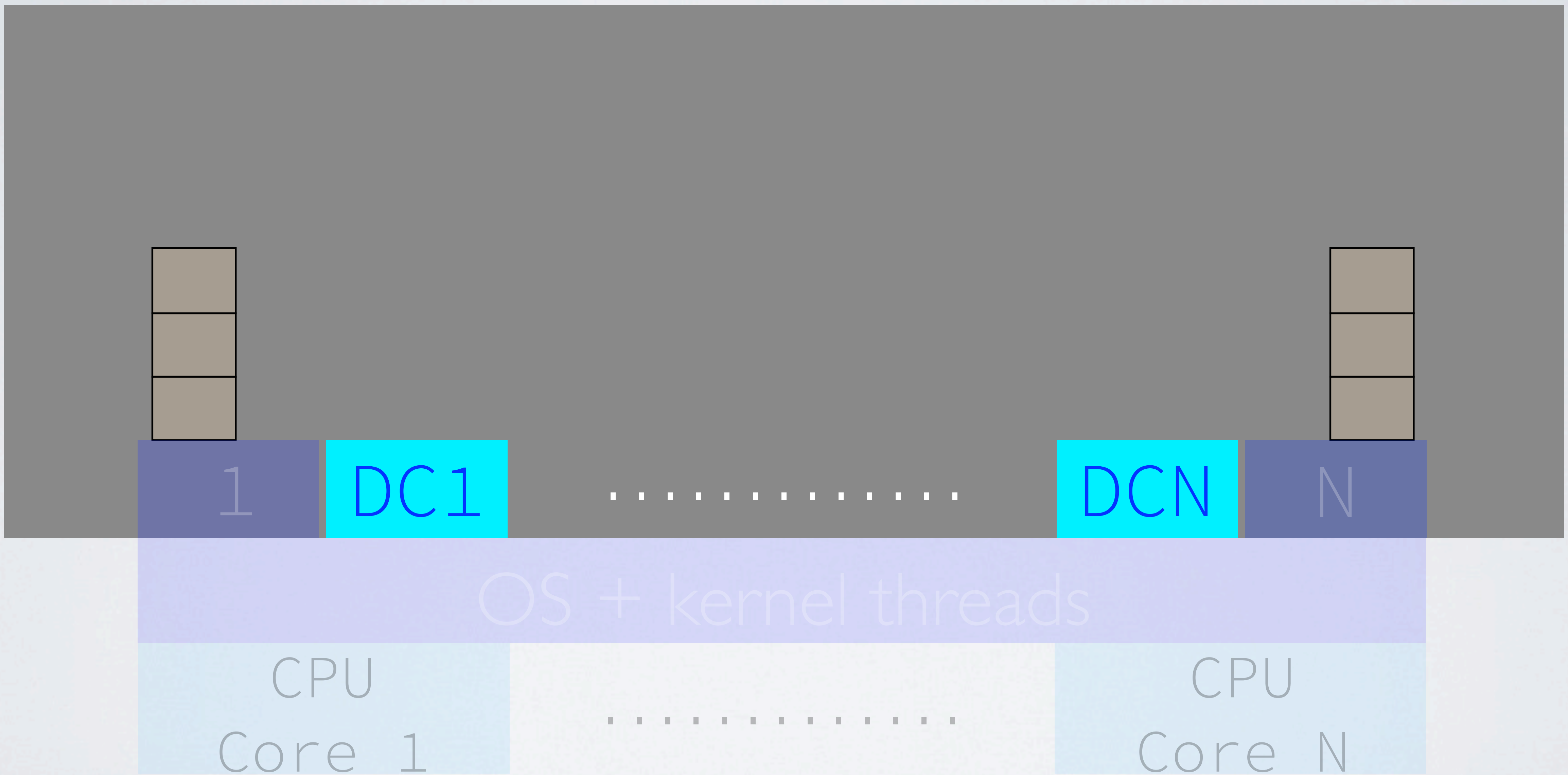
DIRTY SCHEDULERS



DIRTY SCHEDULERS

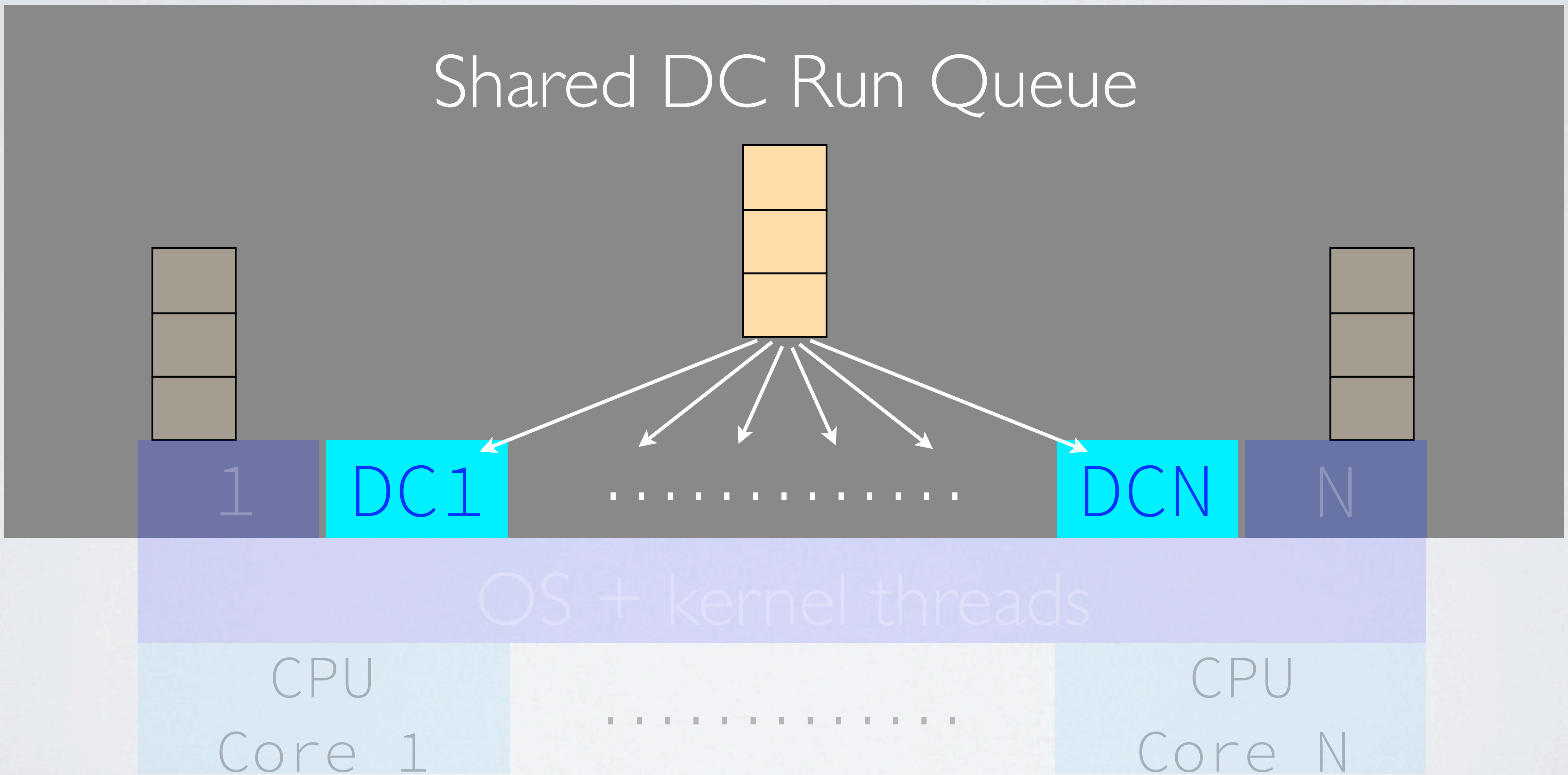


DIRTY SCHEDULERS



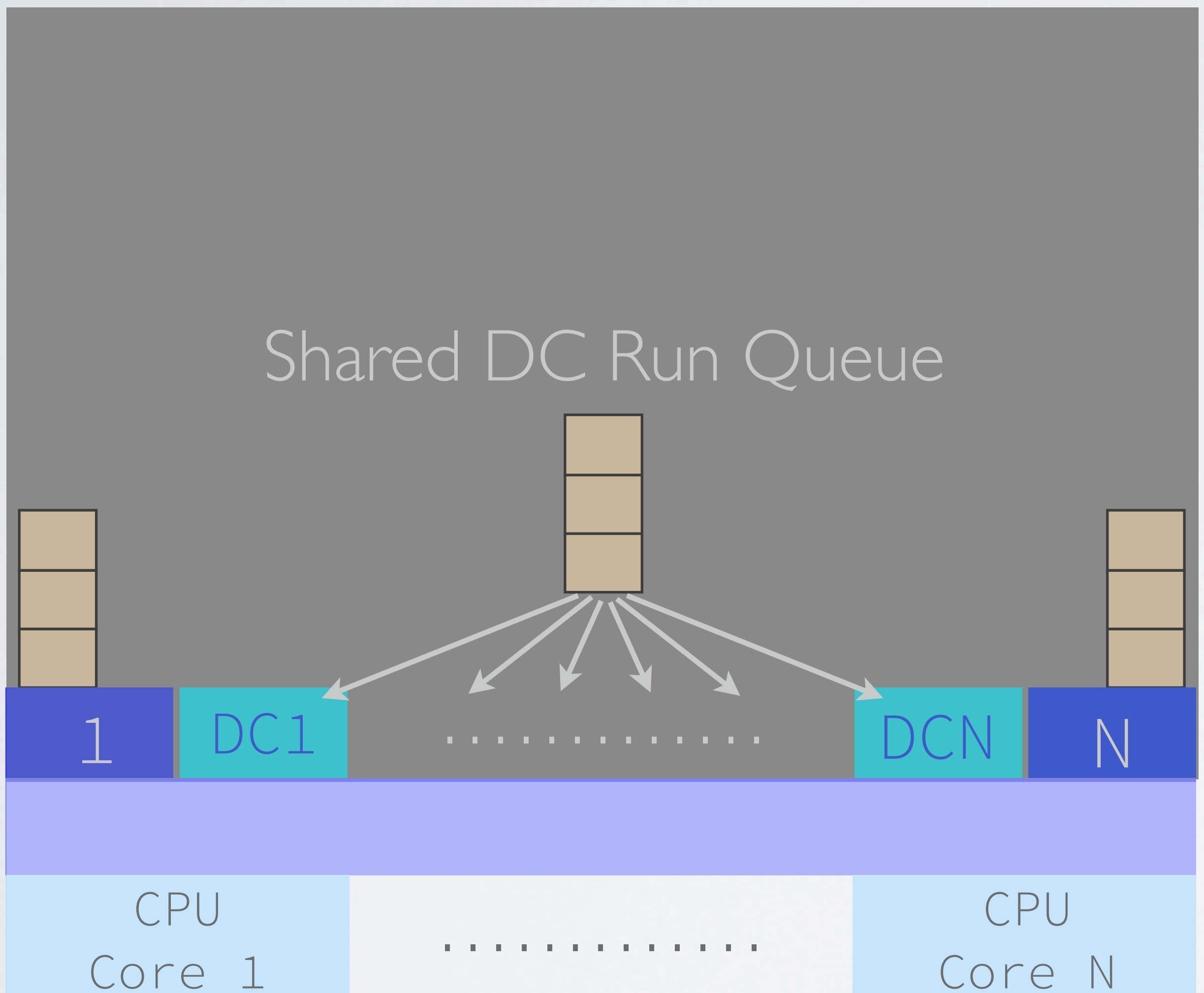
DC: Dirty CPU Scheduler

DIRTY SCHEDULERS

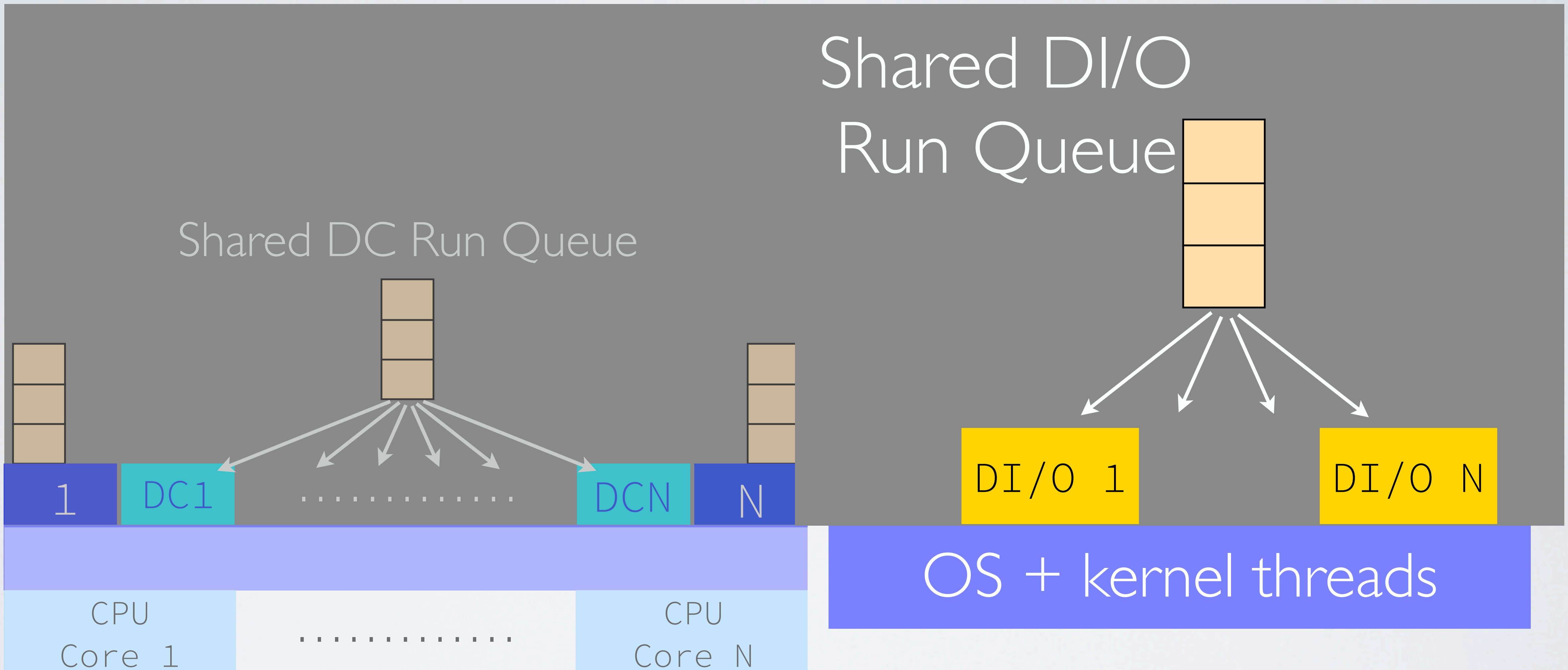


DC: Dirty CPU Scheduler

DIRTY SCHEDULERS

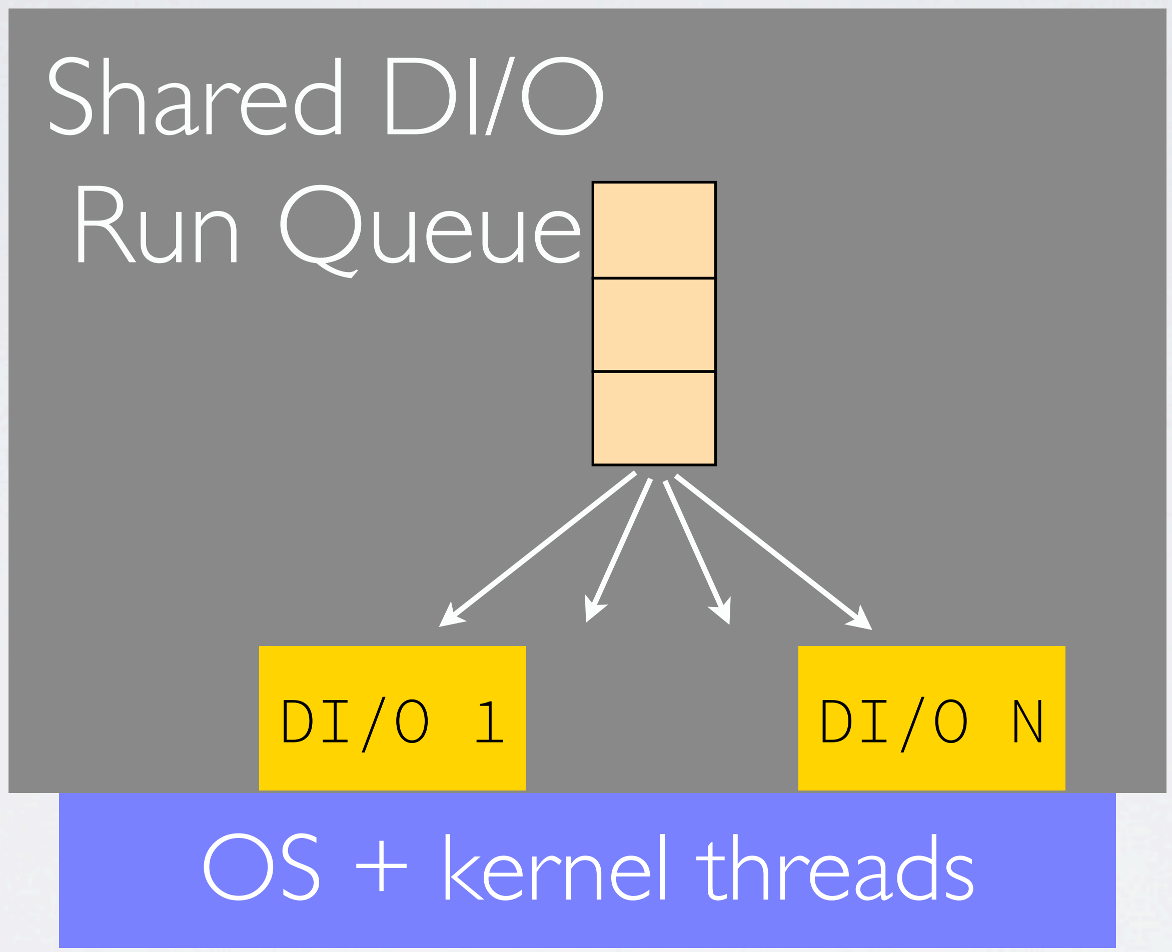


DIRTY SCHEDULERS



DI/O: Dirty I/O Scheduler

DIRTY SCHEDULERS



DI/O: Dirty I/O Scheduler

ENABLING DIRTY SCHEDULERS

- configure --enable-dirty-schedulers
- Your Erlang shell will print something like the following system version line:

  Erlang/OTP 17 [erts-6.2] [source] [64-bit] [smp:8:8] \
  [ds:8:8:10] [async-threads:10] [kernel-poll:false]

USING DIRTY SCHEDULERS

- Either schedule a dirty NIF via **enif_schedule_nif**
  - Pass a flag to indicate dirty CPU or dirty I/O scheduling
- Or specify a NIF as dirty in your **ErlNifFuncs** array
- Both of these are new with Erlang 17.3, replacing old experimental dirty NIF API

USING DIRTY SCHEDULERS

```
static ErlNifFunc funcs[] = {
    {"exor_bad", 2, exor},
    {"exor_yield", 2, exor_yield},
    {"exor_dirty", 2, exor, ERL_NIF_DIRTY_JOB_CPU_BOUND},
};
```

USING DIRTY SCHEDULERS

```
static ErlNifFunc funcs[] = {
    {"exor_bad", 2, exor},
    {"exor_yield", 2, exor_yield},
    {"exor_dirty", 2, exor, ERL_NIF_DIRTY_JOB_CPU_BOUND},
};
```

USING DIRTY SCHEDULERS

```
static ErlNifFunc funcs[] = {
    {"exor_bad", 2, exor},
    {"exor_yield", 2, exor_yield},
    {"exor_dirty", 2, exor, ERL_NIF_DIRTY_JOB_CPU_BOUND},
};
```

A DIRTY EXOR/2

```
7> bitwise:reds(Bin,16#5A,fun bitwise:exor_dirty/2).
{5949862,0,{reductions,5},{reductions,13}}
```

A DIRTY EXOR/2

```
7> bitwise:reds(Bin,16#5A,fun bitwise:exor_dirty/2).
{5949862,0,{reductions,5},{reductions,13}}
```

- 5.95 seconds on a dirty scheduler thread
- 8 reductions and 0 yields
  - But was (almost) never on a regular scheduler
  - Regular schedulers were running other jobs normally

SCHEDULE IT DIRTY

SCHEDULE IT DIRTY

- No chunking or yielding needed for dirty exor/2

SCHEDULE IT DIRTY

- No chunking or yielding needed for dirty exor/2
- But dirty schedulers are finite resources

SCHEDULE IT DIRTY

- No chunking or yielding needed for dirty exor/2
- But dirty schedulers are finite resources
- Evil dirty NIFs can completely occupy all dirty schedulers and prevent other dirty jobs from running

SCHEDULE IT DIRTY

- No chunking or yielding needed for dirty exor/2
- But dirty schedulers are finite resources
- Evil dirty NIFs can completely occupy all dirty schedulers and prevent other dirty jobs from running
- A dirty NIF can use **enif_schedule_nif** to reschedule, yielding to allow other dirty jobs to execute

SCHEDULE IT DIRTY

- No chunking or yielding needed for dirty exor/2
- But dirty schedulers are finite resources
- Evil dirty NIFs can completely occupy all dirty schedulers and prevent other dirty jobs from running
- A dirty NIF can use **enif_schedule_nif** to reschedule, yielding to allow other dirty jobs to execute
- A NIF can use **enif_schedule_nif** to flip itself between regular mode and dirty mode

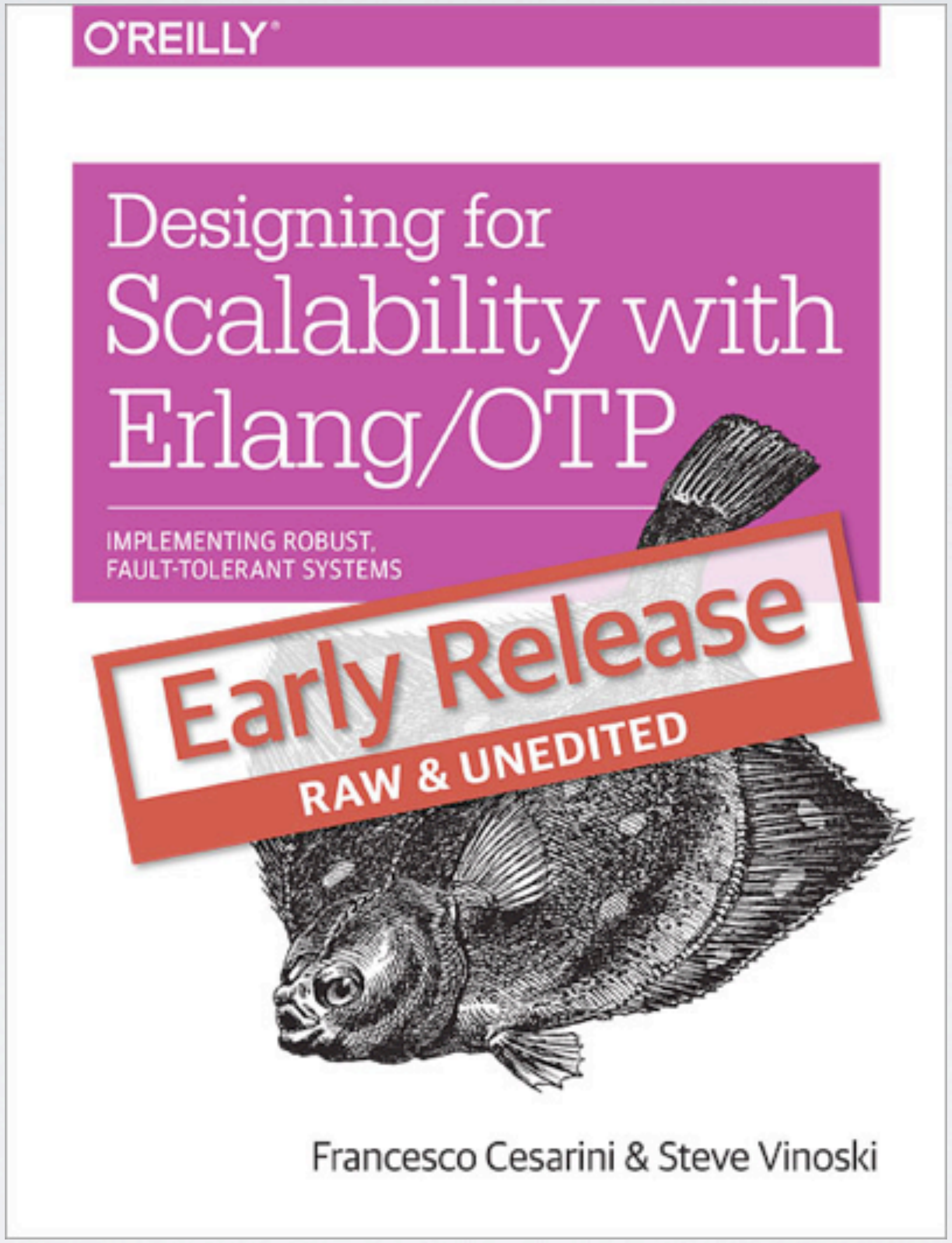
NEXT STEPS

- Dirty drivers already in progress
- Native processes?
  - see Rickard Green's original 2011 presentation on these topics: http://www.erlang-factory.com/upload/presentations/377/RickardGreen-NativeInterface.pdf

ACKNOWLEDGEMENTS

- A huge thanks to Rickard Green of the Ericsson OTP team, who has patiently guided me in this work
- Also thanks to Sverker Eriksson of the OTP team
- And thanks to Anthony Ramine for mentioning "NIF traps" one day in the #erlang IRC channel, where I got the idea for **enif_schedule_nif**

THANKS



http://shop.oreilly.com/product/0636920024149.do#